**Task#1**

**Code:**

#include<iostream>

#include<cstdlib>

#include<ctime>

using namespace std;

struct Node {

int data;

Node\* left;

Node\* right;

Node(int val) :data(val), left(nullptr), right(nullptr) {}

};

class Base {

protected:

Node\* root;

public:

Base() {

root = nullptr;

}

virtual void insert(int val) = 0;

virtual void printInOrder() = 0;

virtual bool find(int val) = 0;

virtual void deleteNode(int val) = 0;

};

class Bst :public Base {

public:

Bst():Base(){}

virtual void insert(int val) {

insertData(val, root);

}

void printInOrder() {

inOrder(root);

cout << endl;

}

bool find(int val) {

return findNode(val, root);

}

void deleteNode(int val) {

root = deleteNodeHelper(root, val);

}

private:

void insertData(int val, Node\*& temp) {

if (temp == nullptr) {

temp = new Node(val);

return;

}

else {

if (val < temp->data) {

insertData(val, temp->left);

}

else if (val > temp->data) {

insertData(val, temp->right);

}

else if (val == temp->data) {

return;

}

}

}

void inOrder(Node\* temp) {

if (temp != nullptr) {

inOrder(temp->left);

cout << temp->data << " ";

inOrder(temp->right);

}

}

bool findNode(int val, Node\* temp) {

if (temp != nullptr) {

if (val == temp->data) {

return true;

}

else if (val < temp->data) {

return findNode(val, temp->left);

}

else if (val > temp->data) {

return findNode(val, temp->right);

}

}

return false;

}

Node\* deleteNodeHelper(Node\* temp, int val) {

if (temp == nullptr) {

return temp;

}

if (val < temp->data) {

temp->left = deleteNodeHelper(temp->left, val);

}

else if (val > temp->data) {

temp->right = deleteNodeHelper(temp->right, val);

}

else {

if (temp->left == nullptr) {

Node\* rightChild = temp->right;

delete temp;

return rightChild;

}

else if (temp->right == nullptr) {

Node\* leftChild = temp->left;

delete temp;

return leftChild;

}

Node\* minLargerNode = findMin(temp->right);

temp->data = minLargerNode->data;

temp->right = deleteNodeHelper(temp->right, minLargerNode->data);

}

return temp;

}

Node\* findMin(Node\* temp) {

while (temp->left != nullptr) {

temp = temp->left;

}

return temp;

}

};

int main() {

srand(time(0));

Bst obj;

int val = 0;

for (int i = 0; i < 7; i++) {

val = rand() % 10;

cout << "Input : " << val << endl;

obj.insert(val);

}

cout << "InOrder : ";

obj.printInOrder();

cout << "Find(5) : " << obj.find(5) << endl;

cout << "Delete(5) : ";

obj.deleteNode(5);

obj.printInOrder();

cout << endl;

return 0;

}

**Output:**

****

**Task#2**

**Code:**

int sumLeafNodes() {

return sumLeafHelper(root);

}

int sumLeafHelper(Node\* temp) {

if (temp == nullptr) {

return 0;

}

if (temp->left == nullptr && temp->right == nullptr) {

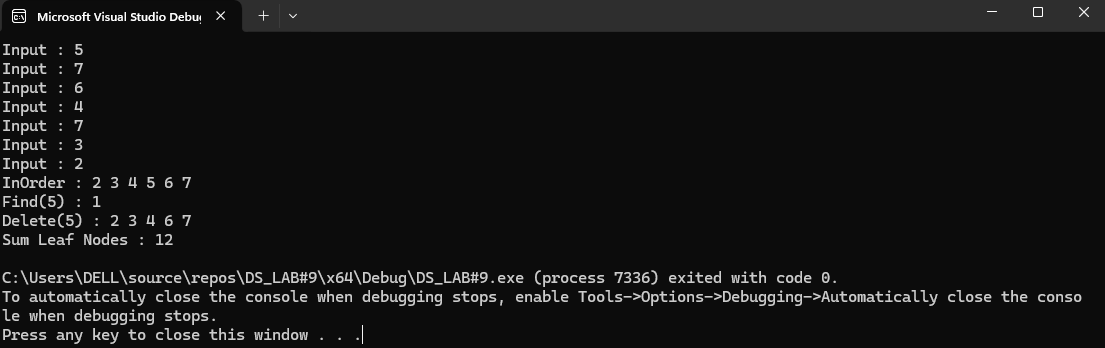
return root->data;

}

return sumLeafHelper(temp->left) + sumLeafHelper(temp->right);

}

**Output:**

****

**Task#3**

**Code:**

void printLevel(int k) {

printLevelHelper(root, k);

}

void deleteLevel(int k) {

deleteLevelHelper(root, k);

}

void printLevelHelper(Node\* temp, int k) {

if (temp == nullptr) {

return;

}

if (k == 0) {

cout << temp->data << " ";

}

else {

printLevelHelper(temp->left, k - 1);

printLevelHelper(temp->right, k - 1);

}

}

void deleteLevelHelper(Node\*& temp, int k) {

if (temp == nullptr) {

return;

}

if (k == 0) {

delete temp;

temp = nullptr;

}

else {

deleteLevelHelper(temp->left, k - 1);

deleteLevelHelper(temp->right, k - 1);

}

}

**Output:**

****

**Task#4**

**Code:**

int closestValue(int key) {

return closestValueHelper(root, key);

}

int closestValueHelper(Node\* root, int key) {

int closest = root->data;

while (root != nullptr) {

if (abs(root->data - key) < abs(closest - key)) {

closest = root->data;

}

if (key < root->data) {

root = root->left;

}

else if (key > root->data) {

root = root->right;

}

else {

break;

}

}

return closest;

}

**Output:**

**A screenshot of a computer

Description automatically generated**

**Task#5**

**Code:**

bool isIdentical(Bst obj1, Bst obj2) {

return isIdenticalHelper(obj1.root, obj2.root);

}

bool isIdenticalHelper(Node\* root1, Node\* root2) {

if (root1 == nullptr && root2 == nullptr) {

return true;

}

if (root1 == nullptr || root2 == nullptr) {

return false;

}

return (root1->data == root2->data) && isIdenticalHelper(root1->left, root2->left) && isIdenticalHelper(root1->right, root2->right);

}

**Output:**

**A screen shot of a computer

Description automatically generated**

**Task#6**

**Code:**

struct NodeEvolutionary {

int data;

NodeEvolutionary\* children[3];

NodeEvolutionary(int val) : data(val) {

for (int i = 0; i < 3; i++) {

children[i] = nullptr;

}

}

};

void createEvolutionaryTree(int rootData, int maxHeight) {

NodeEvolutionary\* rootEvo = new NodeEvolutionary(rootData);

createTree(rootEvo, maxHeight, 0);

}

void createTree(NodeEvolutionary\* node, int maxHeight, int currentHeight) {

if (currentHeight >= maxHeight) {

return;

}

int numChildren = node->data % 4;

for (int i = 0; i < numChildren; i++) {

int childValue;

if (i == 0) {

childValue = node->data + 1;

}

else if (i == 1) {

childValue = node->data + 3;

}

else {

childValue = node->data + 4;

}

node->children[i] = new NodeEvolutionary(childValue);

createTree(node->children[i], maxHeight, currentHeight + 1);

}

}

bool searchInEvolutionaryTree(NodeEvolutionary\* node, int key) {

if (node == nullptr) return false;

if (node->data == key) return true;

for (int i = 0; i < 3; i++) {

if (searchInEvolutionaryTree(node->children[i], key)) {

return true;

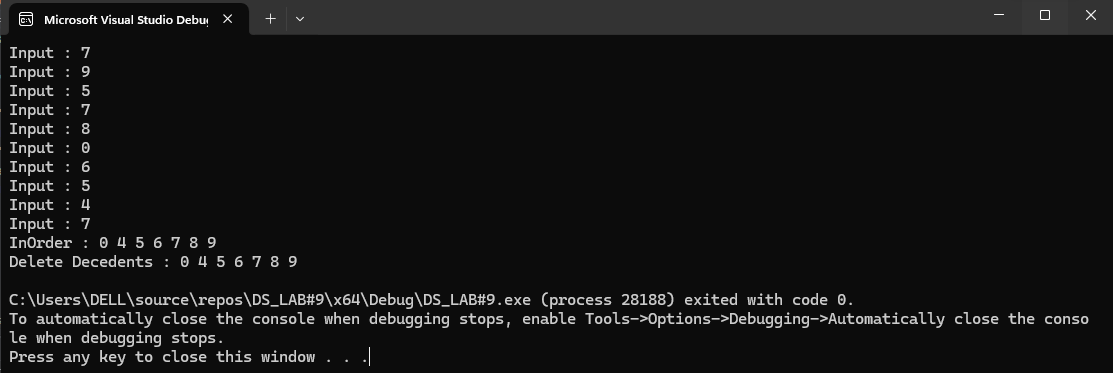
}

}

return false;

}

**Output:**



**Task#7**

**Code:**

void deleteDescendants(int key) {

Node\* nodeToDelete = findNode(root, key);

if (nodeToDelete != nullptr) {

deleteSubtree(nodeToDelete->left);

deleteSubtree(nodeToDelete->right);

nodeToDelete->left = nullptr;

nodeToDelete->right = nullptr;

}

}

Node\* findNode(Node\* node, int key) {

if (node == nullptr) return nullptr;

if (node->data == key) {

return node;

}

if (key < node->data) {

return findNode(node->left, key);

}

else {

return findNode(node->right, key);

}

}

void deleteSubtree(Node\* node) {

if (node == nullptr) return;

deleteSubtree(node->left);

deleteSubtree(node->right);

delete node;

}

**Output:**

**A screenshot of a computer

Description automatically generated**